# A Survey and Classification of A* based Best-First Heuristic Search Algorithms

Luis Henrique Oliveira Rios and Luiz Chaimowicz

Departamento de Ciência da Computação
Universidade Federal de Minas Gerais
{lhrios,chaimo}@dcc.ufmg.br

**Abstract.** A* (*a-star*) is a well known best-first search algorithm that has been applied to the solution of different problems. In recent years, several extensions have been proposed to adapt it and improve its performance in different application scenarios. In this paper, we present a survey and classification of the main extensions to the A* algorithm that have been proposed in the literature. We organize them into five classes according to their objectives and characteristics: *incremental*, *memory-concerned*, *parallel*, *anytime*, and *real-time*. For each class, we discuss its main characteristics and applications and present the most representative algorithms.

## 1 Introduction

Single-agent best-first search algorithms have been broadly applied in the resolution of several problems. One of the most important algorithms in this area is A* (*a-star*) [1]. A* has been used to solve problems of various areas such as the alignment of multiple DNA sequences in biology, path planning in robotics and digital games and classical artificial intelligence problems like the fifteen-puzzle. In spite of being extensively used, A* may present problems in some situations. Firstly, depending on the characteristics of the problem and heuristics used, its cost can be prohibitive. Also, some contexts such as dynamic environments or real-time searches may require adaptations in the original algorithm. Consequently, several extensions to the algorithm have been proposed in the last few years.

This paper presents a survey and classification of the main extensions to the A* algorithm that have been proposed in the literature. Since most of the extensions have specific objectives and try to improve similar points of the original algorithm, this type of classification is important to discuss and compare the new algorithms and to help users and researchers to keep track of the improvements. In this paper, we divide the algorithms into five classes: *incremental*, *memory-concerned*, *parallel*, *anytime*, and *real-time* according to their main characteristics. This paper is organized as follows: we start by presenting a quick review of the A* algorithm. Then, on Section 3, we describe each class discussing its main characteristics and presenting the most representative algorithms. Finally, Section 4 summarizes the classes and presents the conclusion and possibilities for future work.

## 2 Best-First Heuristic Search: A*

A* [1] is probably one of the most well known Artificial Intelligence algorithms. Its objective is to find the shortest path in a graph from a node $x_{\text{start}}$ to a node $x_{\text{goal}}$. As a best-first heuristic search, it employs a function $f$ that guides the selection of the next node that will be expanded. Using the notation from [2], function $f(x)$ is an estimate of $f^*(x)$ that is the cost of the shortest path that passes through a node $x$ and achieves the goal. These two functions are computed as follows: $f(x) = g(x) + h(x)$ and $f^*(x) = g^*(x) + h^*(x)$. The term $g(x)$ is an estimate of $g^*(x)$, the cost of the shortest path from $x_{\text{start}}$ to $x$, and $h(x)$ is an estimate of $h^*(x)$, the cost of the shortest path from $x$ to $x_{\text{goal}}$. If $h(x) \leq h^*(x)$ for all $x$ in the graph the heuristic is admissible and the algorithm can be proved optimal.

The implementation of the A* algorithm generally uses two lists named *open* and *closed*. The *open* list stores the nodes that are in the frontier of the search. The *closed* list stores the nodes that have already been expanded. In each iteration, the algorithm removes the node from the *open* list that has the smallest $f$-value, expands this node (inserts its successors that have not been expanded yet in the *open* list) and marks the node as expanded, inserting it in the *closed* list. It executes these steps until it removes $x_{\text{goal}}$ from the *open* list or until there are no more nodes available in the *open* list. In the first case, A* has computed the shortest path between $x_{\text{start}}$ and $x_{\text{goal}}$. If the second stop condition is true, there are not any solution available.

The main drawback of A* is that the number of nodes expanded can be exponential in the length of the optimal path. Therefore, the number of nodes stored in these two structures can be exponential in the length of the solution. The complexity analysis of A* and related algorithms depends primarily on the quality of the heuristic function and has been the subject of a large body of research, for example [3].

## 3 Classes

As mentioned, in recent years several extensions to the A* algorithm have been proposed. In this section, we organize them into five classes according to their objectives and characteristics: *incremental*, *memory-concerned*, *parallel*, *anytime*, and *real-time*. For each class, we discuss its main characteristics and applications and present the most representative algorithms.

### 3.1 The Incremental Class

The algorithms that belong to the incremental class assume that a series of similar search problems will be performed. The main idea is to reuse information from previous searches to solve new search problems faster than recomputing

each problem from the beginning [4]. This is specially useful in dynamic environments, in which the state space may change between searches. There are basically three different approaches to achieve this reuse of information [5] [1]

- The first type restores the content of A* *open* and *closed* lists to the point in time when the current A* search can diverge from the previous one. That is, the state of an A* search, represented by the content of its lists, is recovered to allow the reuse of the beginning of the immediately preceding search tree. To be faster than a repetition of A* searches, this type of incremental search needs to efficiently reestablish the search state. Examples of algorithms that belong to this type of incremental search are: iA* [7] and Fringe-Saving A* (FSA*) [7]. The experiments executed in [7] showed that FSA* is faster than iA* and can be faster than LPA* (explained below) in some specific situations.
- The second type updates the heuristic (the nodes' $h$-values) to make it more accurate. Algorithms that belong to this class employ the last solution cost and the $g$-value of each node to change the $h$-values. They keep the heuristic consistent after the update phase. The fact that the heuristic is always consistent before starting another search guarantees that they will find an optimal solution if a solution exists. Generalized Adaptive A* (GAA*) [8] is an incremental heuristic search algorithm that adopts this technique to reuse information from previous searches.
- The last type of incremental heuristic search based on A* transforms the search tree of the previous search into the current search tree. Therefore, to be faster than an execution of A* from scratch, the overlap between the old and the new search trees needs to be large. Thus, these algorithms count on the fact that changes on the state space will affect only small branches of the search tree. If the changes occur near the search tree root, these algorithms will not perform well because there will be a small chance of large overlaps between the search trees. Lifelong Planning A* (LPA*) [9], D* [10] and D* Lite [2] are examples of this type of incremental heuristic search.

These algorithms, mainly the ones that belong to the last type, have been largely applied in robotics due to the dynamic fashion of most robotic applications. The D* Lite (a state-of-the-art incremental heuristic search algorithm [5]) is an example of algorithm that has been extensively applied in Robotics. In the same way as A*, D* Lite maintains, for each node $x$, an estimate of $g^*(x)$ denoted by $g(x)$. It also maintains a field named $rhs(x)$ that represents a one step lookahead of $g(x)$. Its value is computed based on $x$ predecessors. According to the relation between these values, a node will be classified as local consistent ($g(x) = rhs(x)$) or local inconsistent (if $g(x) > rhs(x)$ it is overconsistent and if $g(x) < rhs(x)$ it is underconsistent).

[1] Although the authors have reported a problem with the experiments performed in [5] (see [6] for more information), the classification of incremental algorithms suggested on it still be valid.

D* Lite expands the nodes from $x_{\text{goal}}$ to $x_{\text{start}}$. Its expansion loop expands the local inconsistent nodes following a precedence computed for each one using its $g$-value, $rhs$-value and $h$-value. During this phase, its $g$-value and $rhs$-value are modified to eliminate the local inconsistency. D* Lite executes this loop until there are no more inconsistent nodes or until $s_{\text{start}}$ becomes local consistent. The algorithm main loop calls this expansion loop to do the first search and starts executing the plan (moves the agent in direction to $x_{\text{goal}}$). If something in the graph changes, it updates the affected nodes and executes the expansion loop to recompute the shortest path.

Another incremental algorithm is called Field D* [11]. It is strongly based on D* Lite and has been used for path planning of real robotic systems in discretized (grid) environments. Traditional grid techniques may produce unnatural paths and require unnecessary robot movements because they restrict the robots' motion to a small set of directions (for example, 0, $\frac{\pi}{4}$, $\frac{\pi}{2}$, ...). However, this algorithm can generate smooth paths through non-uniform cost grids. That is, Field D* generates paths that can enter and exit cells at arbitrary positions. Basically, it changes the way the nodes are extracted from the grid. It also employs an interpolation to improve the cost estimation. The result are more natural paths in discretized environments.

### 3.2 The Memory-Concerned Class

The algorithms of this class tackle more specifically the problem related with the amount of memory necessary to store the *open* and *closed* lists. As mentioned, one of the main drawbacks of A* is the amount of space necessary to store the *open* and *closed* lists. This space restriction can be unacceptable for some applications. For example, the alignment of multiple DNA or protein sequences can be formalized as a shortest-path problem. A challenging feature of this particular search problem is its large branching factor, which is equal to $2^n - 1$ where $n$ is the number of sequences to be aligned. The increased branching factor significantly worsens the algorithm memory requirements. Therefore, depending on the size of the problem, algorithms like A* can run out of memory before finding a solution.

There are three basic types of memory-concerned algorithms. The first two uses only the main (internal) memory [12], while the third one uses secondary memory as well.

The first type, named memory-efficient, does not keep the *open* and *closed* lists in memory. Typically, this kind of algorithm expands the same node multiple times and its space requirement is linear in the length of the solution. These algorithms clearly work on the traditional computer science trade-off: the compromise between memory usage and execution time. Two examples are Iterative-Deepening A* (IDA*) [13] and Recursive Best-First Search (RBFS) [14]. IDA* was derived from iterative deepening depth-first search. Differently from the original A*, it does not store the lists *open* and *closed*. It executes successive depth-first searches pruning paths that have a $f$-value greater than a threshold. The initial value of this boundary is the $f$-value of the start node. When it is not

possible to proceed with the depth-first search, it updates the boundary to the smaller value that exceeded the previous threshold. The algorithm repeats these steps until it finds the optimal solution. Its main weakness is the expansion of the same nodes multiple times.

One drawback of memory efficient algorithms is that they do not use all the internal memory available, so their running times tend to be longer. Named memory-bounded, the second type of memory concerned algorithms tries to overcome this problem retaining at least one of A* structures and limiting the maximum space occupied by them. For example, SMA* [15] and SMAG* [12] control the growth of the *open* list. When there is no more available space, they prune the least promising nodes to enable the insertion of more nodes. Another algorithm that belongs to the memory-bounded type is the Partial Expansion A* (PEA*) [16]. The idea behind this algorithm is to store only the necessary nodes for finding an optimal solution. PEA* has a predefined constant that helps in the decision of which nodes must be stored in the *open* list. To guarantee the optimal solution, the algorithm stores an additional value for each node that represents the lowest $f$-value among its unpromising child nodes. This algorithm was applied to the multiple sequence alignment problem and, on average, it could align seven sequences with only 4.7% of the amount of memory required by A*.

The third type of memory-concerned algorithms uses the secondary memory besides the main memory to store the algorithm structures. This kind of storage, disks for example, provides much more space with a small cost. However, to efficiently use this space, the algorithms must access it sequentially as the seek time is very high. These algorithms have mainly been applied to solve problems that demands a large amount of memory. The Frontier A*, for example, was implemented to store its *open* list using external memory [17]. Employing a technique called delayed duplicate detection (DDD), it does an external sort with multi-way merge and during this process it removes the duplications. One application of this algorithm, in the 4-peg Towers of Hanoi problem, searched a space with more than one trillion nodes. Frontier A* was also used to the optimal sequence alignment [18] outperforming the best existing competitors.

### 3.3 The Parallel Class

The algorithms of this class are suited for parallel execution, *i.e*, they explore the possibilities of concurrent execution to solve the search problem faster. Depending on the memory architecture, they are characterized as shared memory or distributed memory algorithms.

There are several reports on the adoption of parallel best-first heuristic search algorithms to solve search problems. Most of them were designed for distributed memory architectures. The Parallel Retracting A* (PRA*) [19], for example, has been used to solve the fifteen-puzzle with a significant speedup. PRA* can examine a large number of nodes simultaneously. It maintains an *open* list and a *closed* list for each processor in its local memory. To map the nodes to processors, a hash function is employed. As the first solution found is not necessarily the optimal one (because each processor expands the locally best node), PRA*

maintains a global value with the best solution cost found so far. A similar algorithm, Parallel Local A* (PLA*) [20], has been successfully applied to solve the Traveling Salesman Problem (TSP). Each processor has its local *open* and *closed* lists and the processors interact to inform the best solution found, to redistribute the work, to send or receive cost updates and to detect the algorithm termination.

On the other hand, the Parallel Best-NBlock-First (PBNF) algorithm [21] is an example of a shared memory parallel heuristic search algorithm. The PBNF implementation employs a technique called Parallel Structured Duplicate Detection (PSDD) to avoid the necessity of synchronization for each expanded node. The idea is to create an abstract function (specific for each kind of problem) that maps several nodes of the original graph to a unique node of the abstract graph (denoted as nblock). PBNF maintains an *open* list and a *closed* list for each nblock. When a node in a nblock $b$ is expanded, its successors can only be in nblock $b$ or in its successors. These set of nblocks are called the duplicate detection scope (DDS) of $b$. The abstract graph is used to help the selection of nblocks which DDS are disjoint. These nblocks can be explored in parallel without synchronization. As there are not any synchronization related with the $f$-values explored, the first solution found may not be the optimal. The algorithm will continue the search while there are nodes with $f$-values smaller than the cost of the current solution. The authors evaluated the algorithm in three different domains: STRIPS planning, grid path-finding and fifteen puzzle. They reported a significant speedup for all the evaluated domains.

### 3.4 The Anytime Class

The fourth class of single-agent best-first heuristic search considered in this work is known as anytime. An algorithm belongs to this class if it is able to give a feasible solution, not necessarily the best one, whenever the algorithm is stopped (except during the initial period when the first solution is being calculated) [22]. These algorithms are also expected to return better solutions when more time is provided for execution. After a sufficient execution time, the solution will converge and its quality will not increase anymore. That is, the main characteristic present in anytime algorithms is the trade-off between solution quality and computation time. This feature is very important for some applications as will be discussed next.

There are several reports of the adoption of anytime algorithms to solve search problems. A common feature of some anytime algorithms is the ability to prune the *open* list after the computation of the first solution, providing the necessary upper bound. The Anytime Weighted A* (AWA*) [23] is an example of such algorithm: it reduces the size of the *open* list when it prunes some paths. Similarly to A*, AWA* maintains two lists of nodes: *open* and *closed*. But it uses two evaluation functions, namely $f(x)$ (which is computed in the same way as A*) and $f'(x)$. The main difference between them is that the last is composed of the sum of $g(x)$ and $h'(x)$ (an inadmissible heuristic). This heuristic is computed by the following equation: $h'(x) = w \times h(x)$. That is, an inadmissible heuristic is

acquired by multiplying the admissible heuristic by $w$, where $w \geq 1$ is a weight factor provided by the user. This parameter adjusts the trade off between time and solution quality.

AWA* employs the $f'(x)$ function to select the best node from the *open* list. Before expanding a node, the algorithm checks if it can prune it. To prune a node its $f$-value needs to be greater than the $f$-value of the best solution found so far. The algorithm can expand the same node more than once, like an implementation of A* that uses an admissible (but not consistent) heuristic. The algorithm termination condition is based on $f$-value and that is why it can find the optimal solution. It also provides an error upper bound that is the $f$-value of the best solution already found less the minimum $f$-value in the *open* list.

Another anytime algorithm is the Anytime Repairing A* (ARA*) [24]. It is very similar in operation to AWA*. The authors have successfully applied it for real-time robot path planning. Specifically, the algorithm performance has been evaluated on two applications: planning of robotic arm movements and path planning for mobile robots in outdoor environments. In the first application, they used other anytime algorithms (Anytime A* [25] and a succession of A* searches) to show that ARA* is more efficient. The robot arm application has a huge number of states and demands an anytime algorithm to prune some paths and reduce the computational resources necessary to compute the optimal solution.

Anytime Dynamic A* (AD*) [26] is an algorithm that combines the benefits of anytime and incremental searches. The authors have evaluated the algorithm using a robotic arm in a dynamic environment to show that it can generate better results than D* Lite [2] and ARA* (both algorithms have inspired the creation of AD*). They have also reported the adoption of AD* to plan paths in large partially known environments. In this context, the robot needs to be able to quickly generate solutions when information about the world changes and improve the solution quality without exceeding the time constraints. The problem was modeled as a search over a 4D state space involving position $(x, y)$, orientation and speed. AD* was able to search this space and provide trajectory planning in a real-time fashion.

### 3.5 The Real-Time Class

The real-time class, also known as local search or agent-centered search [27], groups the algorithms that can search in the presence of time constraints. As they search with a limited lookahead, they can not guarantee the optimal solution. They can be considered a special case of anytime search but, as they have special characteristics that differ them from the approaches treated in subsection 3.4, they are presented separately. Typically, real-time search algorithms interleave planning and execution and are used in situations in which executing the plan within the time limits is more important than minimizing the path cost.

The Learning Real-Time A* (LRTA*) [28] is one of the first real-time heuristic search algorithms and inspired the creation of other approaches. It chooses the next state computing the $f$-value of all successors of the current state. In

this case, $f(x)$ is the sum of the cost associated with the edge that connects the two nodes plus $h(x)$, where $x$ is a successor of $s$ (the current state). The next state will be the $x$ with the smallest $f$-value. Before changing the current state, LRTA* updates its heuristic, setting its value to the $f$-value of the next state. The name "learning" comes from the fact that after solving the problem multiple times, the $h$-values will converge to their exact values (that is, the $h$-values will converge to $h^*$-values). It is a real-time algorithm because one can configure the number of nodes considered (that is, the lookahead) in the step that selects the next state according to the time limits of the application.

Real-Time Adaptive A* (RTAA*) [29] is a real-time algorithm that has been applied to goal directed navigation of characters in real-time strategy (RTS) computer games. RTAA* employs the A* algorithm as a subroutine and limits its execution using a lookahead. After A* execution, the algorithm updates the $h$-values using the $g$-value of the best node in the *open* list. It is worth to mention that, due to its interactive fashion, computer games are very good candidates for the use of real-time planning algorithms.

Another suitable area for real-time search algorithms is Robotics. The Moving Target Search (MTS) [30] algorithm, which is a generalization of LRTA*, has been proposed to deal with robotic problems, for example, of a robot pursuing another one. When the target moves, the algorithm updates the $h$-values. The authors have also investigated bidirectional search and proposed an algorithm called Real-Time Bidirectional Search (RTBS). It can be viewed as a cooperative problem solving for uncertain and dynamic situations. Two kinds of RTBS have been proposed: the centralized RTBS (in which the best action is selected considering the two searches) and the decoupled RTBS (in which the two problem solvers make their decisions independently).

## 4 Summary and Conclusion

The A* algorithm is a powerful tool that can be used to solve a large number of different problems. However, its computational cost may be prohibitive for some applications. To alleviate this problem and to adapt it to different application contexts, various extensions of A* have been proposed. In this paper, we presented a survey and classification of the main extensions to the A* algorithm that have been proposed in the literature. We divided the algorithms in five classes: *incremental*, *memory-concerned*, *parallel*, *anytime*, and *real-time*.

The algorithms that belong to the incremental class reuse information from previous searches to solve similar search problems faster than recomputing each problem from the beginning. This type of algorithm has been largely used in Robotics since there is a lot of dynamism caused by sensor uncertainty and/or by the characteristics of the environment. These algorithms can also be useful in digital games where the environment is dynamic.

The memory-concerned class groups the algorithms that tackles the problems related with the amount of memory necessary to store the *open* and *closed* lists. As mentioned, these structures can grown exponentially depending on the ap-

plication. So, memory-concerned algorithms try to limit the amount of memory used during the execution to be able to find a solution without exhausting the available memory.

An algorithm belongs to the parallel class if it tries to explore the advantages of parallel execution. These algorithms can be interesting for any application that has a hardware that provides the necessary conditions. It is currently being explored mainly in problems with large search spaces.

The anytime algorithms can generate a fast, non-optimal solution and then refine it to the optimal if more time is provided. Some applications have applied these algorithms to find a fast solution that will help the algorithm to prune some paths during the subsequent computation that will find the optimal solution. They can also be employed in situations that need quick answers such as digital games and robotics.

The Real-time class groups algorithms that can operate within time constrains. Typically, real-time search algorithms interleave planning and execution. Some digital games demand this kind of algorithm because they need to generate responses fast as they interact with the user in a real-time fashion.

This survey and classification is a first step in comparing these algorithms. We intend to extend the survey, including other algorithms and features that could not be described here due to space limitations. We also intend to experimentally evaluate these algorithms in order to compare the pros and cons of each class under several metrics in different applications. For this, we are planning to use toy problems as well as real applications in the areas of robotics and digital games.

## References


1. Hart, P.E., Nilsson, N.J., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. Systems Science and Cybernetics, IEEE Transactions on **4**(2) (1968) 100–107
2. Koenig, S., Likhachev, M.: Improved fast replanning for robot navigation in unknown terrain. In: Proc. of the Int. Conf. on Robotics and Automation. (2002) 968–975
3. Korf, R.E., Reid, M.: Complexity analysis of admissible heuristic search. In: Proceedings of the National Conference on Artificial Intelligence - AAAI. (1998)
4. Koenig, S., Likhachev, M., Liu, Y., Furcy, D.: Incremental heuristic search in ai. AI Mag. **25**(2) (2004) 99–112
5. Sun, X., Yeoh, W., Koenig, S.: Dynamic fringe-saving a*. In: AAMAS 09. (2009) 891–898
6. Koenig, S.: Dynamic fringe-saving a* (June 2010) Retrieved 7, 2010 from http://idm-lab.org/bib/abstracts/Koen09e.html.
7. Sun, X., Koenig, S.: The fringe-saving a* search algorithm - a feasibility study. In: IJCAI. (2007) 2391–2397
8. Sun, X., Koenig, S., Yeoh, W.: Generalized adaptive a*. In: AAMAS '08, Int. Foundation for Autonomous Agents and Multiagent Systems (2008) 469–476
9. Koenig, S., Likhachev, M., Furcy, D.: Lifelong planning a*. Artif. Intell. **155**(1-2) (2004) 93–146

10. Stentz, A.: Optimal and efficient path planning for partially-known environments. In: Proc. of the IEEE Int. Conf. on Robotics and Automation. (1994) 3310–3317
11. Ferguson, D., Stentz, A.: Field d*: An interpolation-based path planner and replanner. In: Proc. of the Int. Symposium on Robotics Research. (2005) 1926–1931
12. Zhou, R., Hansen, E.A.: Memory-bounded a* graph search. In: Proc. of the Fifteenth Int. Florida Artif. Intell. Research Society Conf., AAAI Press (2002) 203–209
13. Korf, R.E.: Depth-first iterative-deepening: An optimal admissible tree search. Artif. Intell. **27** (1985) 97–109
14. Korf, R.E.: Linear-space best-first search. Artif. Intell. **62**(1) (1993) 41–78
15. Russell, S.: Efficient memory-bounded search methods. In: In ECAI-92, Wiley (1992) 1–5
16. Yoshizumi, T., Miura, T., Ishida, T.: A* with partial expansion for large branching factor problems. In: Proc. of the Seventeenth National Conf. on Artif. Intell. and Twelfth Conf. on Innovative Applications of Artif. Intell. (2000) 923–929
17. Korf, R.E.: Best-first frontier search with delayed duplicate detection. In: AAAI, AAAI Press / The MIT Press (2004) 650–657
18. Korf, R.E., Zhang, W., Thayer, I., Hohwald, H.: Frontier search. J. ACM **52**(5) (2005) 715–748
19. Evett, M., Hendler, J., Mahanti, A., Nau, D.: Pra*: massively parallel heuristic search. Technical report (1991)
20. Dutt, S., Mahapatra, N.R.: Parallel A* algorithms and their performance on hypercube multiprocessors. In: Proc. of the 7th Int. Parallel Processing Symposium, IEEE Computer Society Press (1993) 797–803
21. Burns, E., Lemons, S., Zhou, R., Ruml, W.: Best-first heuristic search for multicore machines. In: IJCAI. (2009) 449–455
22. Teije, A., Harmelen, F.: Describing problem solving methods using anytime performance profiles. In: Proc. of the 14th European Conf. on Artif. Intell., IOS Press (2000) 181–185
23. Hansen, E.A., Zhou, R.: Anytime heuristic search. J. Artif. Intell. Res. (JAIR) **28** (2007) 267–297
24. Likhachev, M., Gordon, G., Thrun, S.: Ara*: Anytime a* with provable bounds on sub-optimality. In: NIPS 03, MIT Press (2004)
25. Zhou, R., Hansen, E.A.: Multiple sequence alignment using anytime a*. In: Eighteenth National Conf. on Artif. Intell., American Association for Artif. Intell. (2002) 975–976
26. Likhachev, M., Ferguson, D.I., Gordon, G.J., Stentz, A., Thrun, S.: Anytime dynamic a*: An anytime, replanning algorithm. In: ICAPS 2005, AAAI (2005) 262–271
27. Koenig, S.: Agent-centered search. AI Mag. **22**(4) (2001) 109–131
28. Korf, R.E.: Real-time heuristic search. Artif. Intell. **42**(2-3) (1990) 189–211
29. Koenig, S., Likhachev, M.: Real-time adaptive a*. In: AAMAS '06, ACM (2006) 281–288
30. Ishida, T.: Real-time search for autonomous agents and multiagent systems. Autonomous Agents and Multi-Agent Systems **1**(2) (1998) 139–167